インターネット計測とデータ解析 第 8 回

長 健二朗

2015 年 6 月 8 日

前回のおさらい

第 7 回 多変量解析 (6/1)

- データセンシング
- 地理的位置情報 (geo-location)
- 線形回帰
- 主成分分析
- 演習: 線形回帰、主成分分析

今日のテーマ

第 8 回 時系列データ

- インターネットと時刻
- ネットワークタイムプロトコル
- 時系列解析
- 演習: 時系列解析
- 課題 **2**

計測と時間

- 絶対時刻
  - 協定世界時 UTC (Universal Coordinated Time)
    - セシウム原子時計をもとに取り決められている標準時
- 相対時刻
  - 時刻の差分
- 時刻調整
  - 時計の時刻は前後に補正される
  - NTP では 128ms 未満の誤差は一度に、それ以上だと徐々に修正

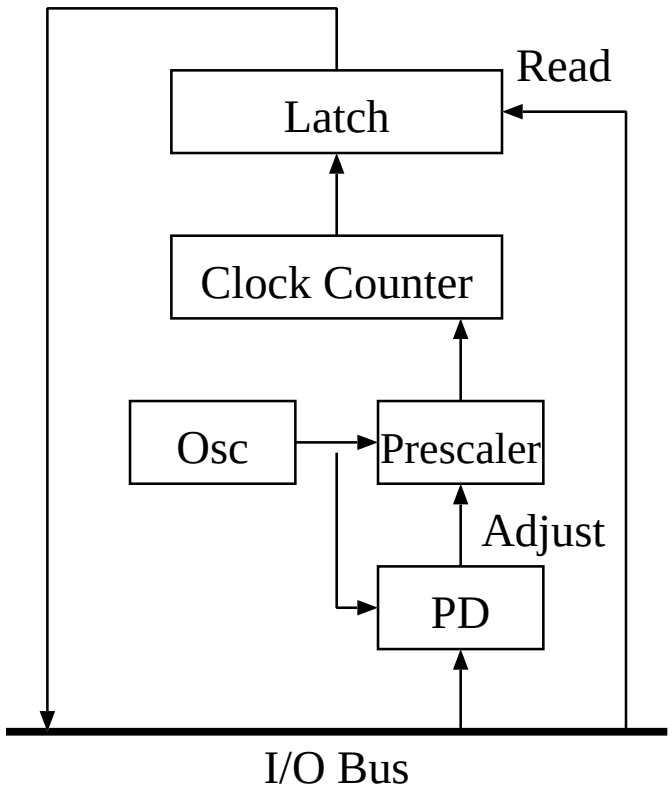
クロックの誤差

- クロックの誤差
  - 同期
    - 2 つのクロックの差
  - 正確さ
    - UTC からのずれ
  - 解像度
    - クロックの精度
  - スキュー
    - 時間とともに同期や正確さがずれる
- 時間粒度
  - PC クロック: 0.1-1sec/日ぐらいずれる
  - NTP: 10-100ms の正確さにクロックを同期
  - tcpdump などのタイムスタンプ:
    - 100usec-100msec (通常 $< 1msec$ だが保証なし)

PC のクロック

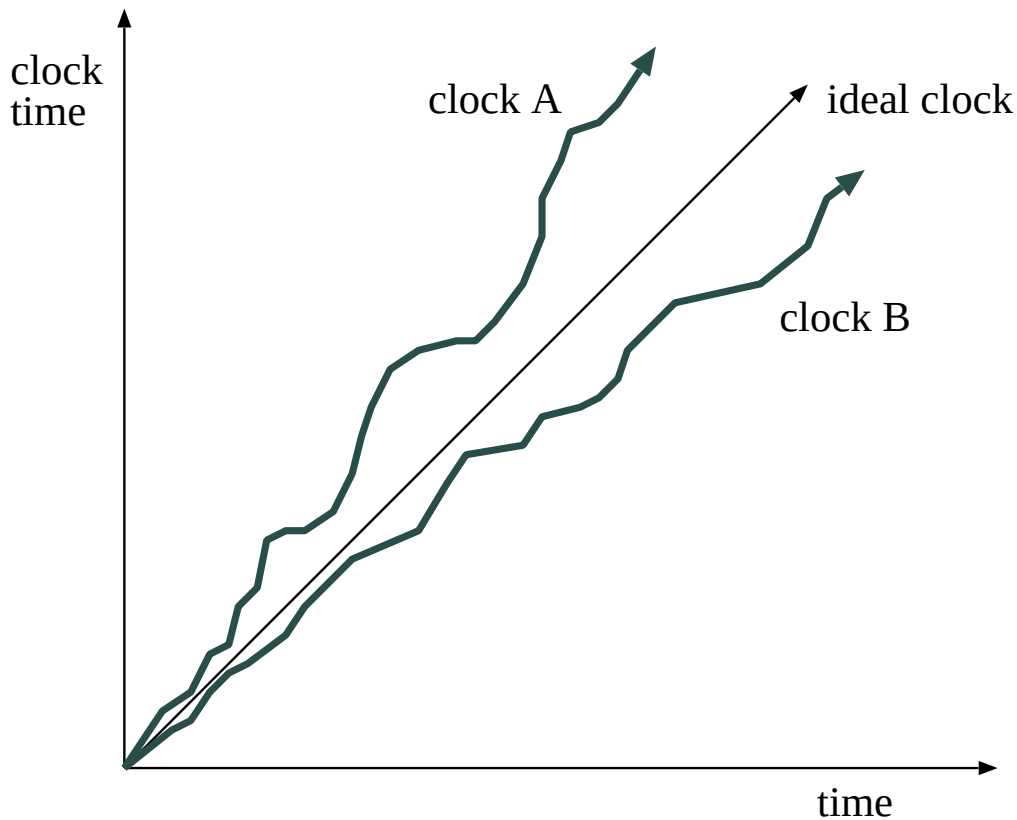
i8254 プログラムインターバルタイマー

- 16-bit フリーラニング ダウンカウンター
  - 1,193,182 Hz の水晶発振器を基にしている
  - カウンターがゼロになると割り込み信号を上げてカンターレジスタ値をリロード



クロックドリフト

- 水晶発振器のドリフト
  - ハードウエア仕様の許容誤差: 10^{-5}
    - 0.86 sec/day は許容誤差内
  - ドリフトは温度に大きく影響される



その他の PC クロック

- Pentium TSC (Time Stamp Counter)
  - CPU クロックで駆動される CPU 内蔵フリーラニングカウンター
  - 可変クロックやマルチ CPU で問題
- ACPI (Advanced Counfiguration and Power Interface)
  - パワー管理機能が提供するフリーラニングカウンター
- Local APIC (Advanced Programmable Interrupt Controller)
  - 各プロセッサに内蔵される割り込み機能付きタイマー
- HPET (High Precision Event Timer)
  - IA-PC の新しいタイマー仕様
  - 2005 年頃からチップセットに組み込み
- 外部クロック
  - GPS、CDMA など時刻情報を含む
    - インターフィスにより読み込みオーバーヘッド

OS 時刻管理

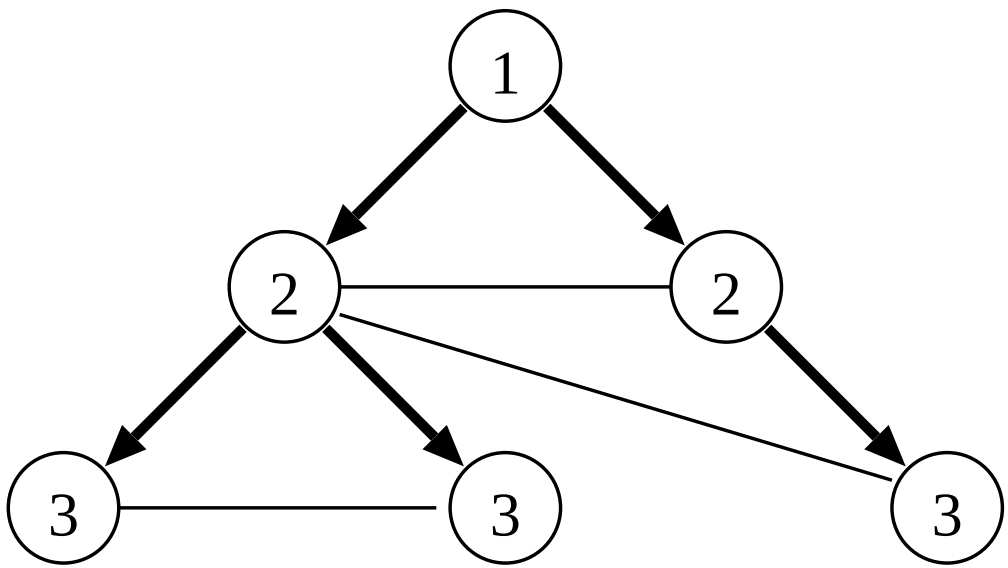
- OS はソフトウエアにより時刻を管理
  - 起動時にカレンダーチップから時刻を得る
  - ハードウエアクロック割り込み毎に時刻をアップデート
- 従来の UNIX では、デフォルトで 10ms ごとにクロック割り込みが発生するようにクロックカウンターを設定

UNIX gettimeofday

- 古い OS ではクロック割り込みの粒度しかなかった
- いまどきの OS ではより高精度の時刻を得られる
  - クロックカウンター値を読み出してソフトウエアクロックを補間
    - i8254 の解像度: 838ns (1 / 1193182)
  - OS 内部処理時間
    - i8254 レジスタアクセス: 1-10usec
    - struct timeval への変換: 1-100usec
  - ユーザ空間から OS 内部へのアクセス
    - システムコール オーバーヘッド: 10-500usec
    - プロセススケジューリングの影響: 1-100msec or more
- タイマーイベント ソフトウエア処理時間 (e.g., setitimer):
  - ソフトウエアタイマー割り込みから処理 (10msec by default)
  - プロセススケジューリングの影響を受ける

NTP (Network Time Protocol)

- インターネット上の複数サーバー間で時刻同期
  - プライマリサーバ: 直接 UTC ソースに繋がる
  - セカンダリサーバ: プライマリに同期
  - 3 段目以降のサーバ: セカンダリ以降に同期
- スケーラビリティ
  - 20-30 プライマリ、 2000 セカンダリを $< 30ms$ に同期
- さまざまな機能
  - 耐故障性、認証などをサポート



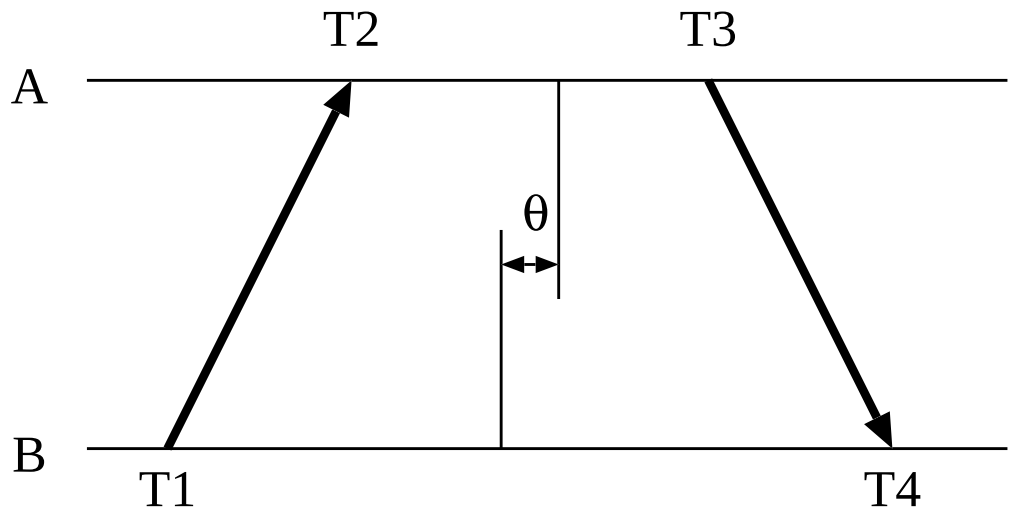
NTP 同期モード

- マルチキャスト (LAN 向け)
  - 定期的に時刻情報をマルチキャストで広報
- リモートプロシージャコール
  - クライアントが (複数) サーバーに時刻情報を要求
- ピアプロトコル
  - 複数のピアの間で同期

NTP ピアプロトコル

相手とのオフセットと通信遅延を計測

- $a = T2 - T1 \qquad b = T3 - T4$
- clock offset: $\theta = (a + b)/2$ (RTT が対称だと仮定)
- roundtrip delay: $\delta = a - b$

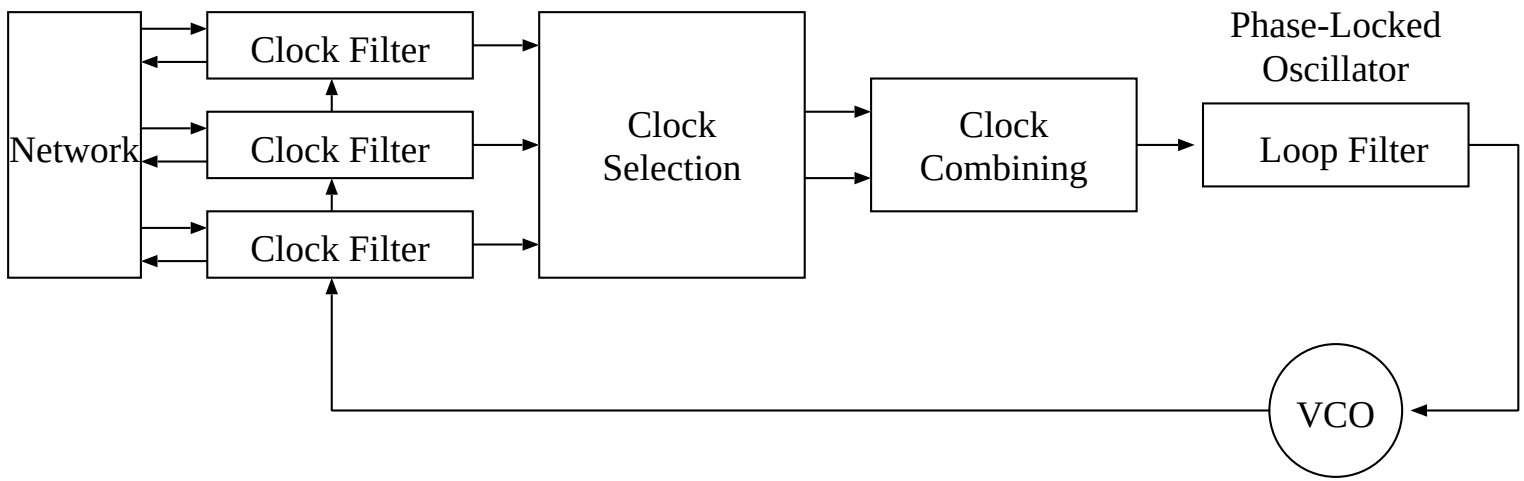


全てのメッセージに以下を含める

- T3: send time (current time)
- T2: receive time
- T1: send time in received message

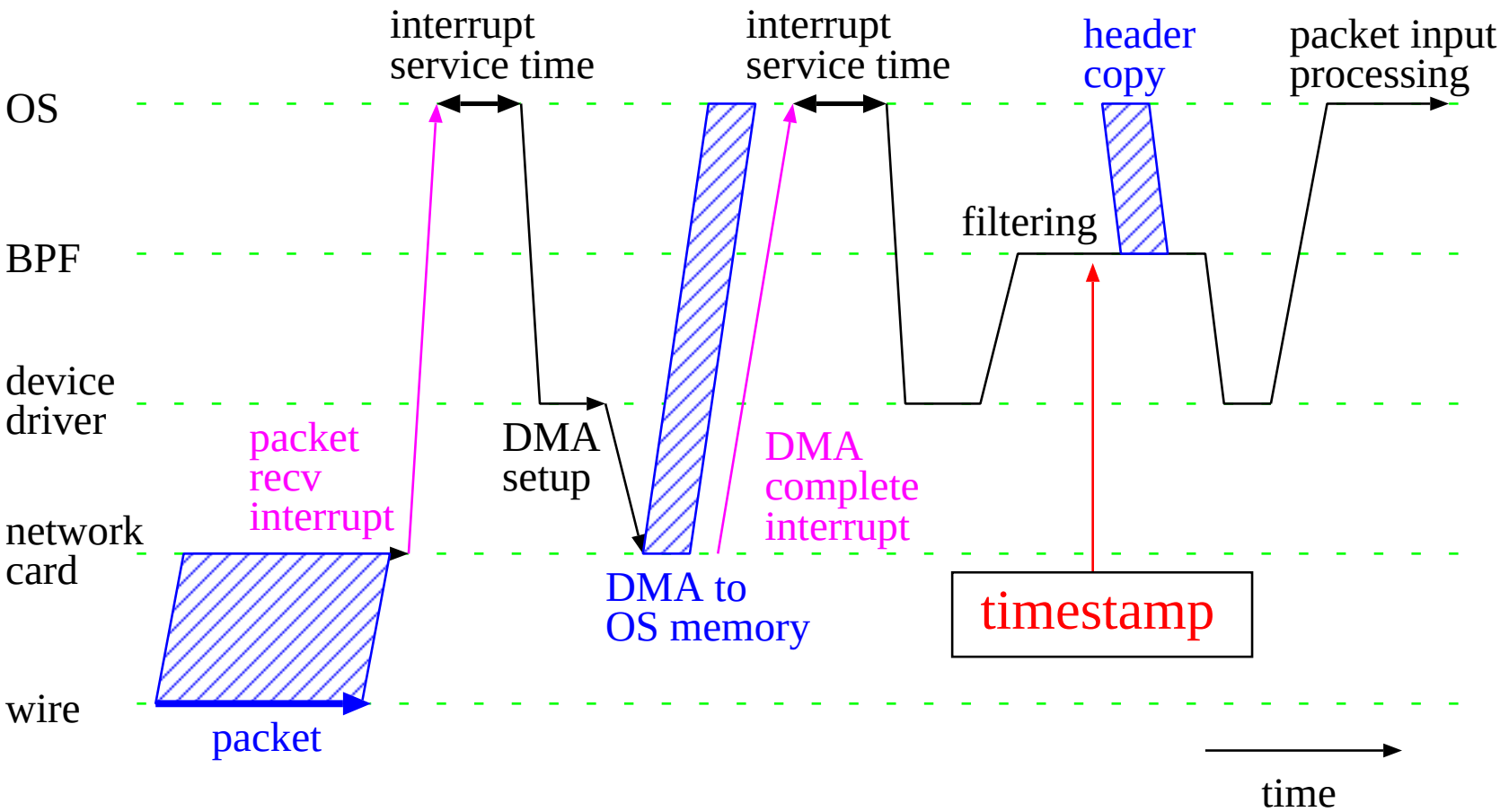
NTP システムモデル

- クロックフィルタ
  - 各ピアからの時刻情報を時系列に平滑化
- クロック選択
  - 互いに一致しているクロックを抜き出す
  - インターセクションアルゴリズム: 外れ値の除外
  - クラスタリング: 最善値の選択
- クロック統合
  - 推定値を 1 個に統合



BSD UNIX の BPF タイムスタンプ

- 通常、割り込み処理 2 回の後タイムスタンプ
  - recv packet, DMA complete



ネットワークトラフィックの時系列解析

時間とともに変化する動的な挙動の解析

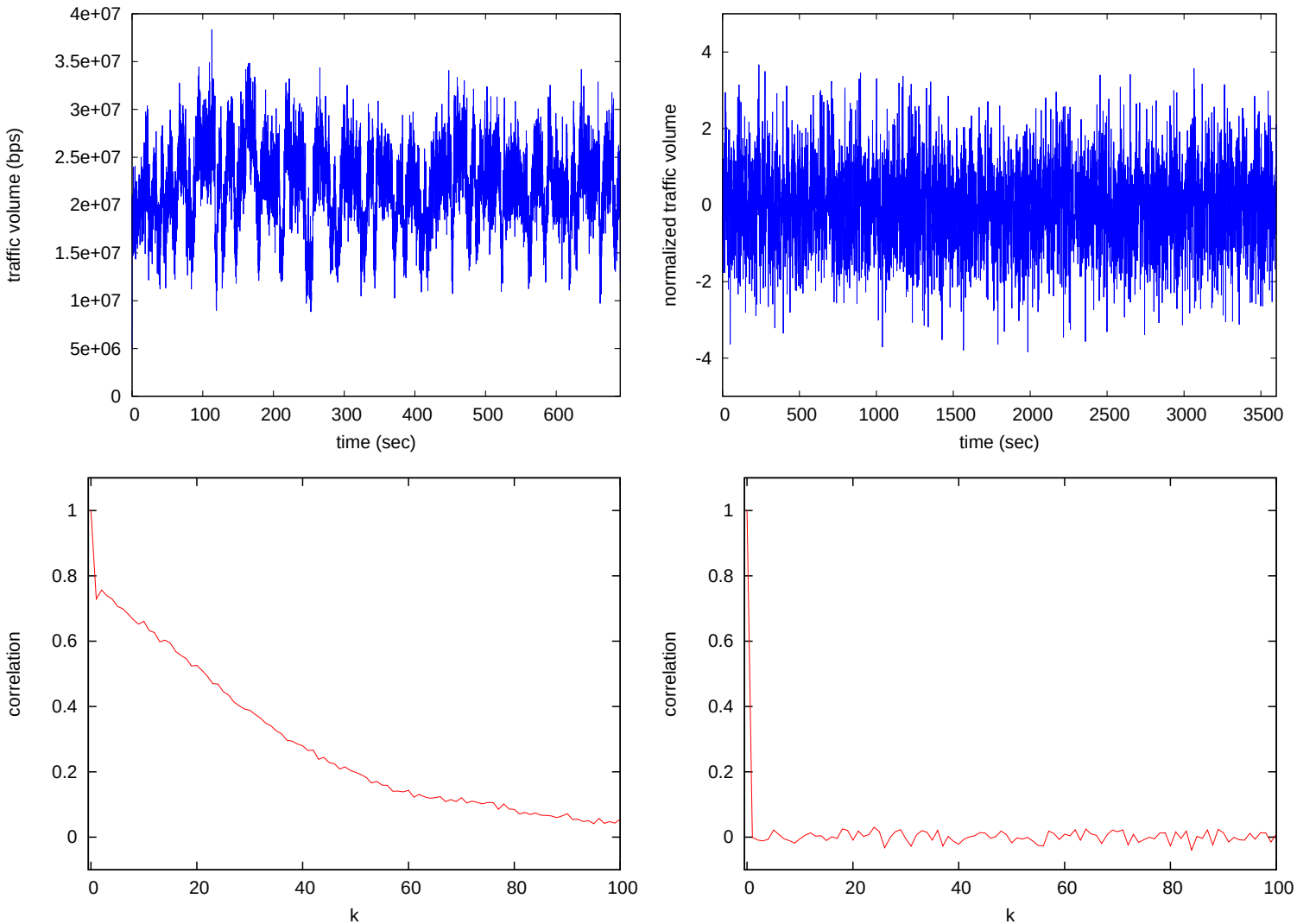
- 数学的な取り扱いは難しい
- 限られたツール

トピック

- 自己相関 (autocorrelation)
- 定常過程 (stationary process)
- 長期記憶 (long-range dependence)
- 自己相似トラフィック (self-similar traffic)

ネットワークトラフィックの自己相関

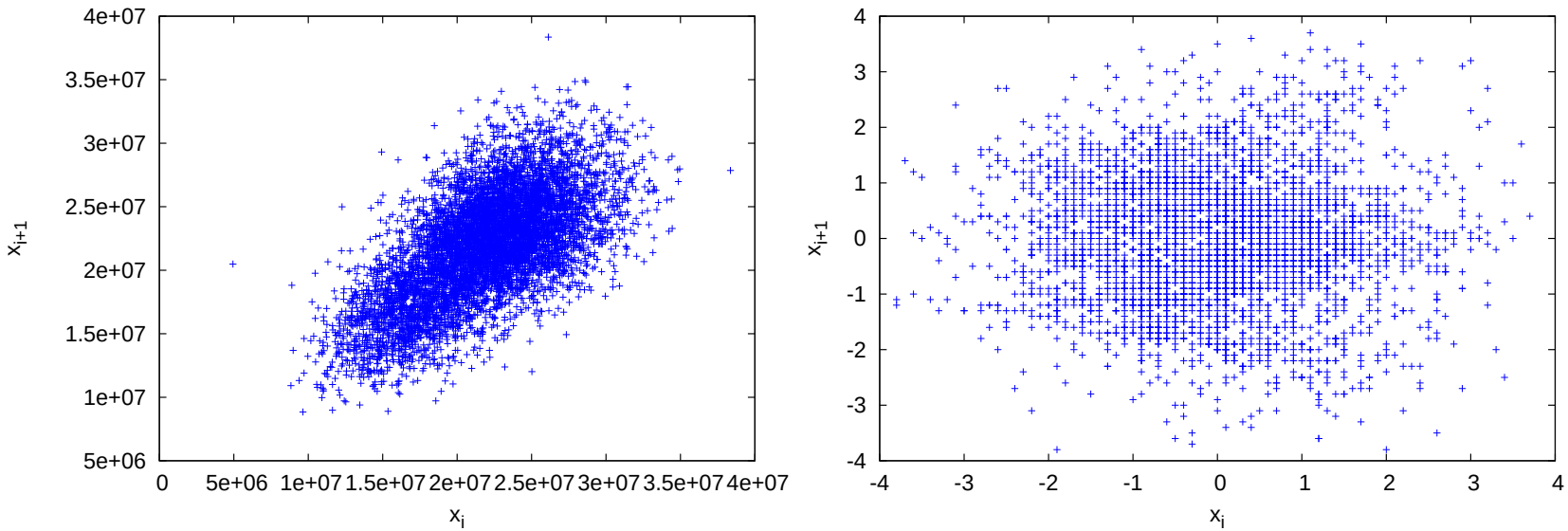
- 過去の状態の影響 (トレンド) と周期性 (日、週、季節)
- 自己相関 (autocorrelation): 同一変数の異なる時間の値の相関



(左) 実トラフィック (右) 乱数から生成したトラフィック (上) 時系列グラフ (下) 自己相関

自己相関とラグプロット

- ラグ (lag) プロット: x_i と x_{i+k} の散布図
  - 自己相関の存在を確認する簡単な方法
  - k を大きくすると長周期の繰り返しパターンを発見可能



ラグプロットの例: (左) 実トラフィック (右) 乱数から生成したトラフィック

自己相関

- 確率過程 (stochastic process)

$$\{x(t), t \in T\}$$

- 自己相関 (autocorrelation): 同一変数の時刻 t_1 の値と t_2 の値の相関
- 自己相関関数 (autocorrelation function)

$$R(t_1, t_2) = E[x(t_1)x(t_2)]$$

- 自己共分散 (autocovariance)

$$Cov(t_1, t_2) = E((x(t_1) - \mu_{t_1})(x(t_2) - \mu_{t_2})] = E[x(t_1)x(t_2)] - \mu_{t_1}\mu_{t_2}$$

定常過程 (stationary process)

- 時系列 X_t が定常過程
  - 平均が変化しない: $E(X_t) = \mu$
  - かつ自己共分散が k にのみ依存

$$\gamma_k = Cov(X_t, X_{t+k}) = E((X_t - \mu)(X_{t+k} - \mu))$$

$$\gamma_0 = Var(X_t) = E((X_t - \mu)^2)$$

- 自己相関係数 (autocorrelation coefficient)
  - 自己共分散を分散で正規化
  - 過去からの影響を示す

$$\rho_k = \frac{\gamma_k}{\gamma_0}$$

ホワイトノイズ

ホワイトノイズ: 定常過程で自己相関係数が 0

$$\rho_k = 0 \ (k \neq 0)$$

IID 過程 (independent identically distributed process)

- 平均と分散が一定のホワイトノイズ
  - 確率過程の話に必ず出てくる
- X_t が互いに独立で同じ分布に従う
  - independent: X_t が互いに独立 (無相関)
  - identically distributed: X_t が同じ分布に従う

非定常過程

- 非定常
  - 平均または自己共分散が時間とともに変化
- 数学的な扱いが困難
  - 一般には時系列の差分を取って定常化する必要
- 定常判定
  - パワースペクトル密度を調べ
    - べき指数が 1.0 より大きい場合は非定常
- ネットワークでは非定常なトラフィックが観測される
  - 輻輳、DoS/flooding 等の攻撃

パワースペクトル密度 (power spectral dencity)

- 定常過程のパワースペクトル密度は自己相関関数のフーリエ変換
  - 時間領域から周波数領域への変換
  - 時系列データを sin,cos の重ね合わせで表現

$$S(f) = \int_{-\infty}^{\infty} R(\tau) e^{-2\pi i f \tau} d\tau$$

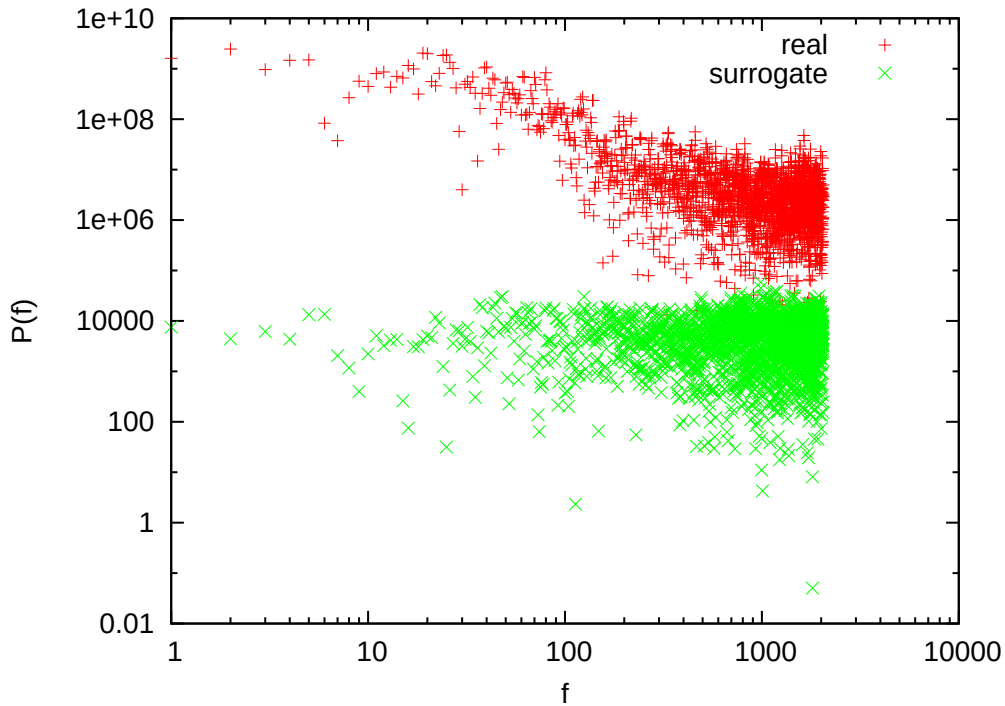
  - パワースペクトル密度

$$P(f) \equiv |S(f)|^2 + |S(-f)|^2,\ 0 \leq f < \infty$$

- パワースペクトル密度は各周波数成分の平均パワーを示す

パワースペクトル密度の性質

- ホワイトノイズ (無相関): $P(f) \sim const$
- 自己相似 (長期記憶): $P(f) \sim f^{-\alpha}, 0 < \alpha \leq 1.0$
- 1/f ゆらぎ (パワーが周波数に反比例): $\alpha = 1.0$
- 非定常: $\alpha > 1.0$



例: (赤) 実トラフィック (緑) 乱数から生成したトラフィック

短期記憶と長期記憶

自己共分散は各々の時差 k の影響を個別に示す。
全体を見るために全ての時差 k について自己共分散の総和を取る

- 短期記憶性
  - $\sum_k \rho(k)$ が有限

$$\sum_{k=0}^{\infty} |\rho(k)| < \infty$$

  - $\rho(k)$ が指数関数と同様か、より早く減衰
  - 特徴
    - 平均値周辺でゆらぐ
    - 遠い過去の影響はない
- 長期記憶性
  - $\sum_k \rho(k)$ が発散

$$\sum_{k=0}^{\infty} |\rho(k)| = \infty$$

  - 自己相関係数が双曲線的に減衰
  - 特徴
    - 平均から大きく外れた値が観測される

自己相似トラフィック

ネットワークトラフィックは厳密な自己相似ではないが、場合によって他より良いモデルを与える

- スケールフリー
- 長期記憶
- 自己共分散がべき的に減衰

$$\rho(k) \sim k^{-\alpha} \quad (k \to \infty) \quad 0 < \alpha < 1$$

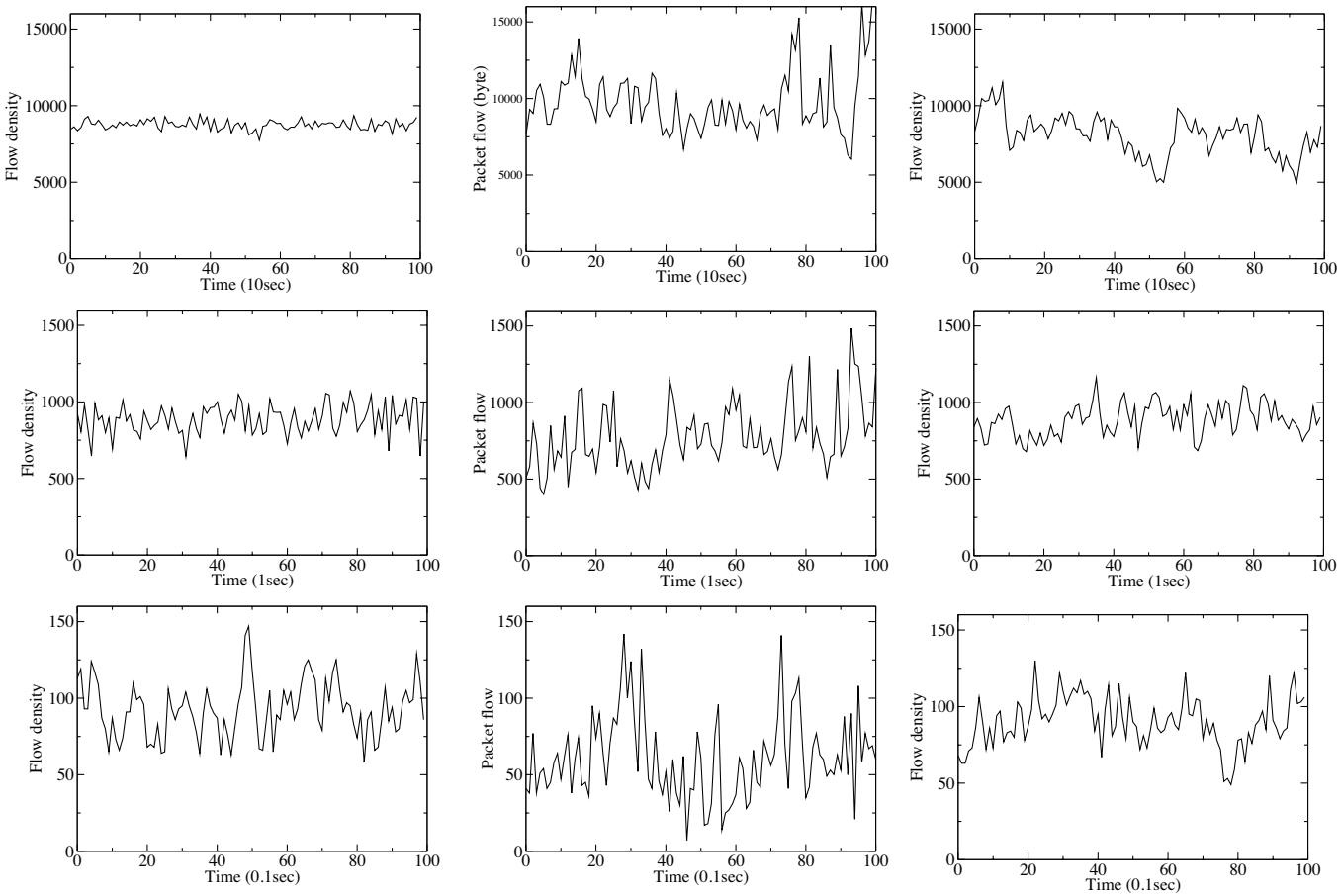
- 同様にパワースペクトル密度もべき的に減衰
  - 低周波成分 (遠い過去) の影響が大きい

$$P(f) \sim |f|^{-\alpha} \quad (f \to 0)$$

- 分散が発散

ネットワークトラフィックの自己相似性

- (左) 指数関数モデル (中) 実トラフィック (右) 自己相似モデル
- 時間粒度: (上)10sec (中)1 sec (下)0.1 sec



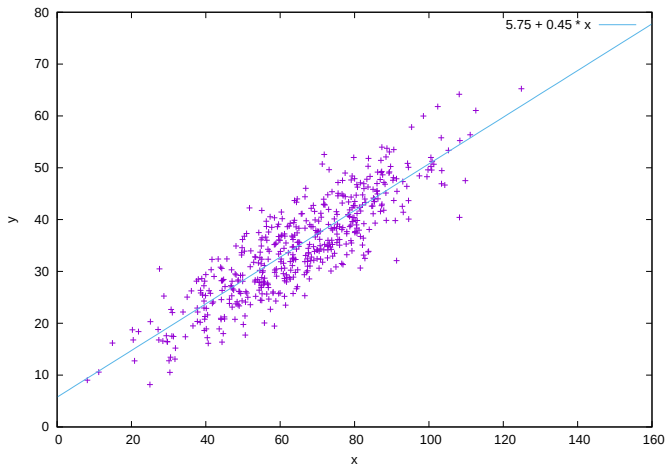
前回の演習: 線形回帰の計算

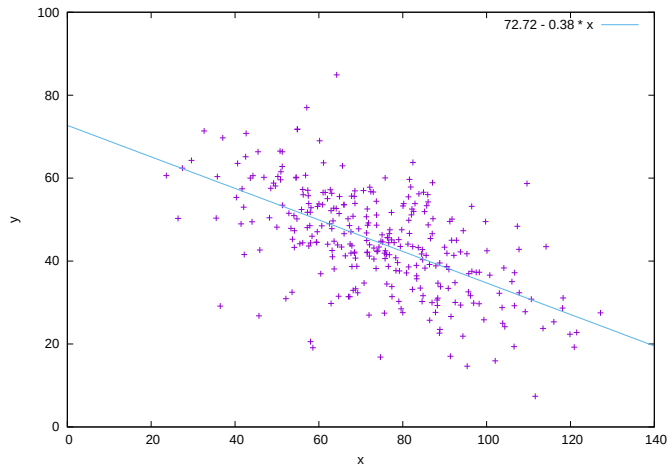
- 前回のデータを使い回帰直線を計算する
  - correlation-data-1.txt, correlation-data-2.txt

$$f(x) = b_0 + b_1 x$$

$$b_1 = \frac{\sum xy - n\bar{x}\bar{y}}{\sum x^2 - n(\bar{x})^2}$$

$$b_0 = \bar{y} - b_1 \bar{x}$$





data-1:r=0.87 (left), data-2:r=-0.60 (right)

演習: 回帰直線の計算スクリプト

```
#!/usr/bin/env ruby

# regular expression for matching 2 floating numbers
re = /([-+]?\d+(?:\.\d+)?)\s+([-+]?\d+(?:\.\d+)?)/

sum_x = sum_y = sum_xx = sum_xy = 0.0
n = 0
ARGF.each_line do |line|
    if re.match(line)
      x = $1.to_f
      y = $2.to_f

      sum_x += x
      sum_y += y
      sum_xx += x**2
      sum_xy += x * y
      n += 1
    end
end

mean_x = Float(sum_x) / n
mean_y = Float(sum_y) / n
b1 = (sum_xy - n * mean_x * mean_y) / (sum_xx - n * mean_x**2)
b0 = mean_y - b1 * mean_x

printf "b0:%.3f b1:%.3f\n", b0, b1
```

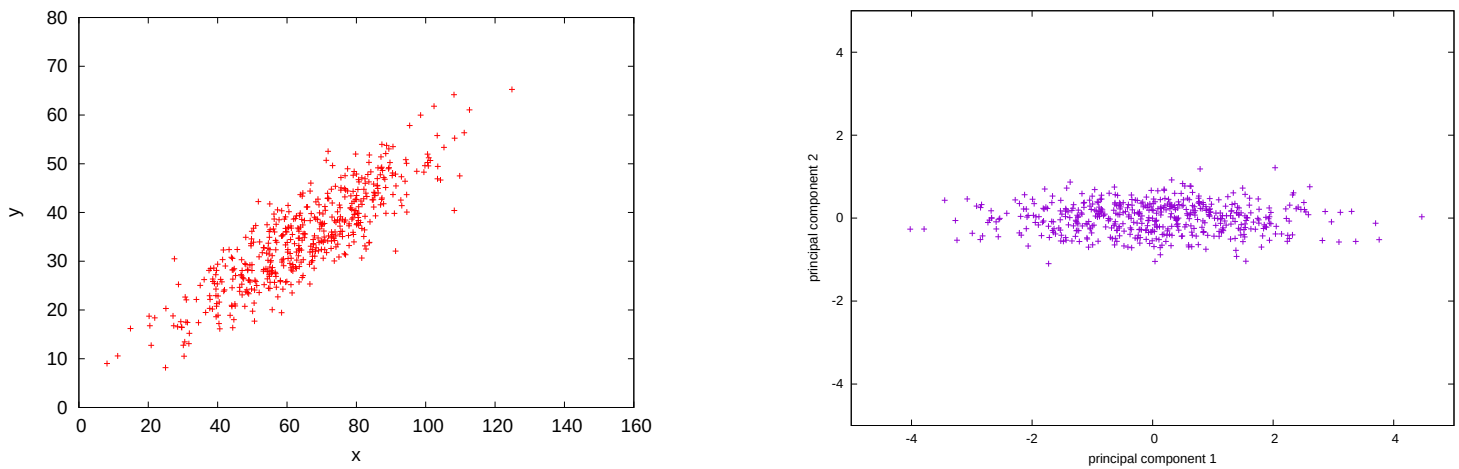
演習: 散布図に回帰直線を加える

```
set xrange [0:160]
set yrange [0:80]

set xlabel "x"
set ylabel "y"

plot "correlation-data-1.txt" notitle with points, \
5.75 + 0.45 * x
```

前回の演習 2: 主成分分析

- 主成分分析: 線形回帰に使ったデータで実行

```
$ ruby pca.rb correlation-data-1.txt
PC1:
eigenval: 1.86477
proportion: 0.93239
cumulative proportion: 0.93239
eigenvector: [0.7071067811865475, 0.7071067811865475]

PC2:
eigenval: 0.13523
proportion: 0.06761
cumulative proportion: 1.00000
eigenvector: [0.7071067811865475, -0.7071067811865475]
```



data-1:r=0.87 (left), pca plot (right)

主成分分析: 3 変数の場合

```
$ cat pca-data.txt
7 4 3
4 1 8
6 3 5
8 6 1
8 5 7
7 2 9
5 3 3
9 5 8
7 4 5
8 2 2
$ ruby pca.rb -p pca-data.txt
-0.542660  0.664959   0.035324
2.803897   -0.066207  0.348792
0.615631   0.306325   0.165059
-2.158526  0.958839   0.386086
-0.931052  -1.044819  0.360013
1.142388   -1.273946  0.471245
0.803082   1.261879   0.472342
-1.246820  -1.655638  -0.023007
-0.286027  -0.024512  0.186799
-0.199912  0.873118   -1.460164
```

```
$ ruby pca.rb pca-data.txt
PC1:
eigenval: 1.76877
proportion: 0.58959
cumulative proportion: 0.58959
eigenvector: [-0.642004576349, -0.686361641360, 0.341669169247]

PC2:
eigenval: 0.92708
proportion: 0.30903
cumulative proportion: 0.89862
eigenvector: [-0.384672291688, -0.0971303301343, -0.917928606687]

PC3:
eigenval: 0.30415
proportion: 0.10138
cumulative proportion: 1.00000
eigenvector: [-0.663217424343, 0.720745028589, 0.20166618906]
```

演習: PCA スクリプト (1/4)

```
#!/usr/bin/env ruby
#
# usage: pca.rb [-p] datafile
# input datafile: row: variables, column: observations
# -p: convert input into principal components

# use matrix class for eigen vector computation
require 'matrix'
require 'optparse'

# nomarlize an array of array (m x n) into bb (m x n)
def normalize_matrix(aa)
  m = aa[0].length
  n = aa.length
  bb = Array.new(n) { Array.new(m) } # normalized array of array

  for i in (0 .. m - 1)
    sum = 0.0 # sum of data
    sqsum = 0.0 # sum of squares
    for j in (0 .. n - 1)
      v = aa[j][i]
      sum += v
      sqsum += v**2
    end
    mean = sum / n
    stddev = Math.sqrt(sqsum / n - mean**2)
    for j in (0 .. n - 1)
      bb[j][i] = (aa[j][i] - mean) / stddev # normalize
    end
  end
  bb # return the new array of array
end
```

演習: PCA スクリプト (2/4)

```
# make correlation matrix from data (array of array)
def make_corr_matrix(aa)
  m = aa[0].length
  n = aa.length
  corr_matrix = Array.new(m) { Array.new(m) }

  for i in (0 .. m - 1)
    for j in (0 .. m - 1)
      sum_x = 0.0
      sum_y = 0.0
      sum_xx = 0.0
      sum_yy = 0.0
      sum_xy = 0.0
      for k in (0 .. n - 1)
        x = aa[k][i]
        y = aa[k][j]
        sum_x += x
        sum_y += y
        sum_xx += x**2
        sum_yy += y**2
        sum_xy += x*y
      end
      cc = 0.0
      denom = (sum_xx - sum_x**2 / n) * (sum_yy - sum_y**2 / n)
      if denom != 0.0
        cc = (sum_xy - sum_x * sum_y / n) / Math.sqrt(denom)
      end
      corr_matrix[i][j] = cc
    end
  end
  corr_matrix
end
```

演習: PCA スクリプト (3/4)

```
do_projection = false
OptionParser.new {|opt|
  opt.on('-p') {|v| do_projection = true}
  opt.parse!(ARGV)
}

# read data into input (array of array)
input = Array.new
ARGF.each_line do |line|
  values = line.split
  if values.length > 0
    row = Array.new
    values.each do |v|
      row.push v.to_f
    end
    input.push row
  end
end

corr_aa = make_corr_matrix(input) # create correlation matrix
corrmatrix = Matrix.rows(corr_aa) # convert array of array into matrix class

# compute the eigenvalues and eigenvectors
# eigensystem returns v: eigenvectors, d: diagonal matrix of eigenvalues,
#  v_inv: transposed matrix of v.  corrmatrix = v * d * v_inv
v, d, v_inv = corrmatrix.eigensystem

# returned vectors are sorted in increasing order of eigenvals.
# so, re-order eigenvalues and eigenvectors in decreasing order
eigenvals = Array.new # array of eigenvalues
(d.column_size - 1).downto(0) do |i|
  eigenvals.push d[i,i]
end
eigenvectors = Matrix.columns(v.column_vectors.reverse)
```

演習: PCA スクリプト (4/4)

```
if do_projection != true
  # show summaries
  eig_sum = 0.0
  eigenvals.each do |val|
    eig_sum += val
  end
  cum = 0.0  # cumulative of eigenvalues
  eigenvals.each_with_index do |val, i|
    printf "PC%d:\n", i + 1
    printf "eigenval: %.5f\n", val
    printf "proportion: %.5f\n", val / eig_sum
    cum += val
    printf "cumulative proportion: %.5f\n", cum / eig_sum
    print "eigenvector: "
    print eigenvectors.column(i).to_a
    print "\n\n"
  end
else
  # project the input into new coordinate
  # first, normalize the input and then convert it to matrix
  normalized = Matrix.rows(normalize_matrix(input))
  # projected data = eigenvec.T x normalized.T
  projected = eigenvectors.transpose * normalized.transpose

  projected.column_vectors.each do |vec|
    vec.each do |v|
      printf "%.6f\t", v
    end
    print "\n"
  end
end
```

今回の演習: 自己相関

- 1 週間分のトラフィックデータから自己相関を計算する

```
# ruby autocorr.rb autocorr_5min_data.txt > autocorr.txt
# head -10 autocorr_5min_data.txt
2011-02-28T00:00 247 6954152
2011-02-28T00:05 420 49037677
2011-02-28T00:10 231 4741972
2011-02-28T00:15 159 1879326
2011-02-28T00:20 290 39202691
2011-02-28T00:25 249 39809905
2011-02-28T00:30 188 37954270
2011-02-28T00:35 192 7613788
2011-02-28T00:40 102 2182421
2011-02-28T00:45 172 1511718
# head -10 autocorr.txt
0 1.000
1 0.860
2 0.860
3 0.857
4 0.857
5 0.854
6 0.851
7 0.849
8 0.846
9 0.841
```

自己相関関数の求め方

タイムラグ k の自己相関関数

$$R(k) = \frac{1}{n}\sum_{i=1}^{n} x_i x_{i+k}$$

$k = 0$ の場合は、同一データの相関なので、$R(k)/R(0)$ で規格化する

$$R(0) = \frac{1}{n}\sum_{i=1}^{n} x_i^2$$

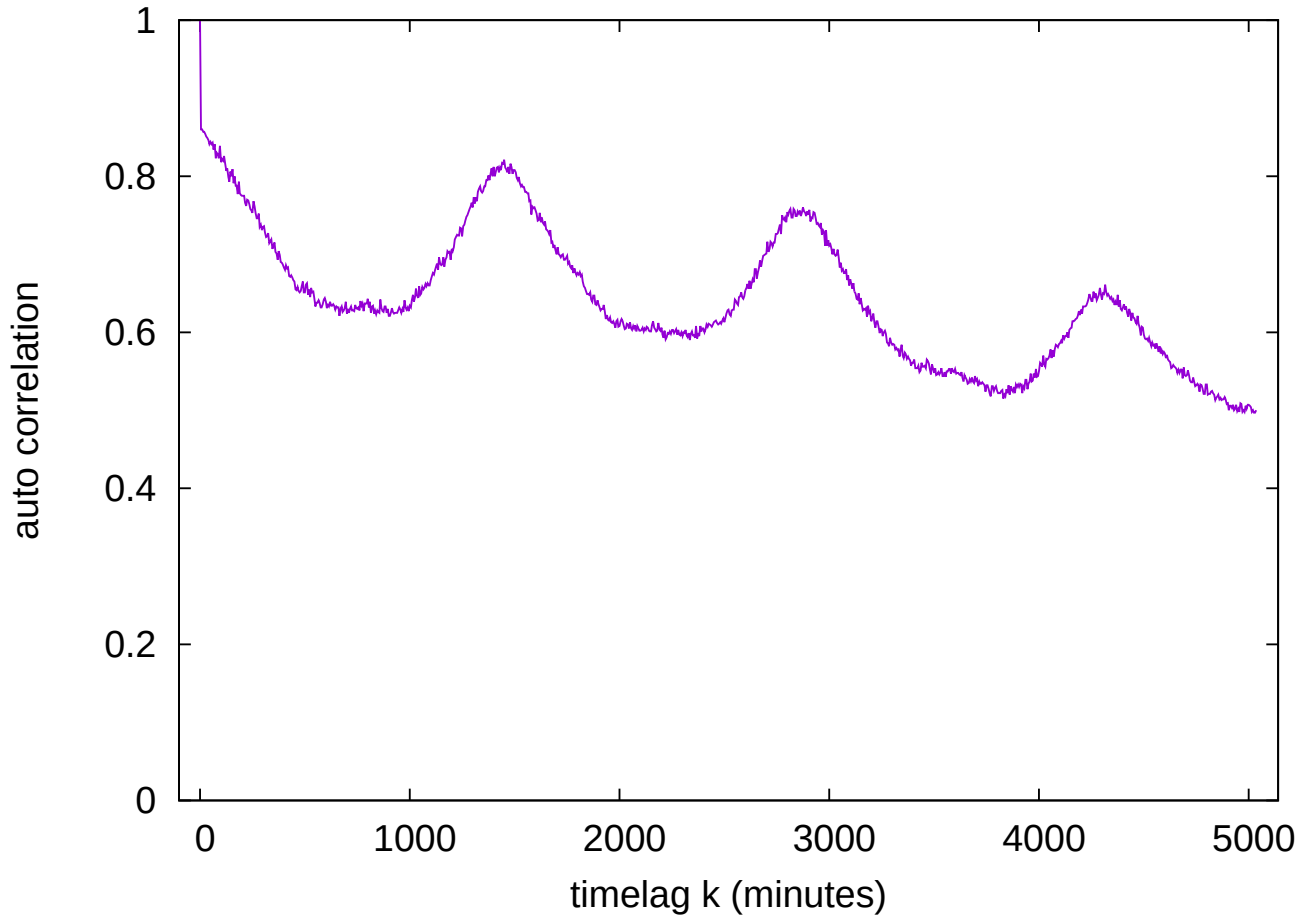
2n 個のデータ数が必要

自己相関関数スクリプト

```
# regular expression for matching 5-min timeseries
re = /^(\d{4}-\d{2}-\d{2})T(\d{2}:\d{2})\s+(\d+)\s+(\d+)/

v = Array.new() # array for timeseries
ARGF.each_line do |line|
  if re.match(line)
    v.push $3.to_f
  end
end

n = v.length # n: number of samples
h = n / 2 - 1 # (half of n) - 1

r = Array.new(n/2) # array for auto correlation
for k in 0 .. h # for different timelag
  s = 0
  for i in 0 .. h
    s += v[i] * v[i + k]
  end
  r[k] = Float(s)
end

# normalize by dividing by r0
if r[0] != 0.0
  r0 = r[0]
  for k in 0 .. h
    r[k] = r[k] / r0
    printf "%d %.3f\n", k, r[k]
  end
end
```

自己相関プロット

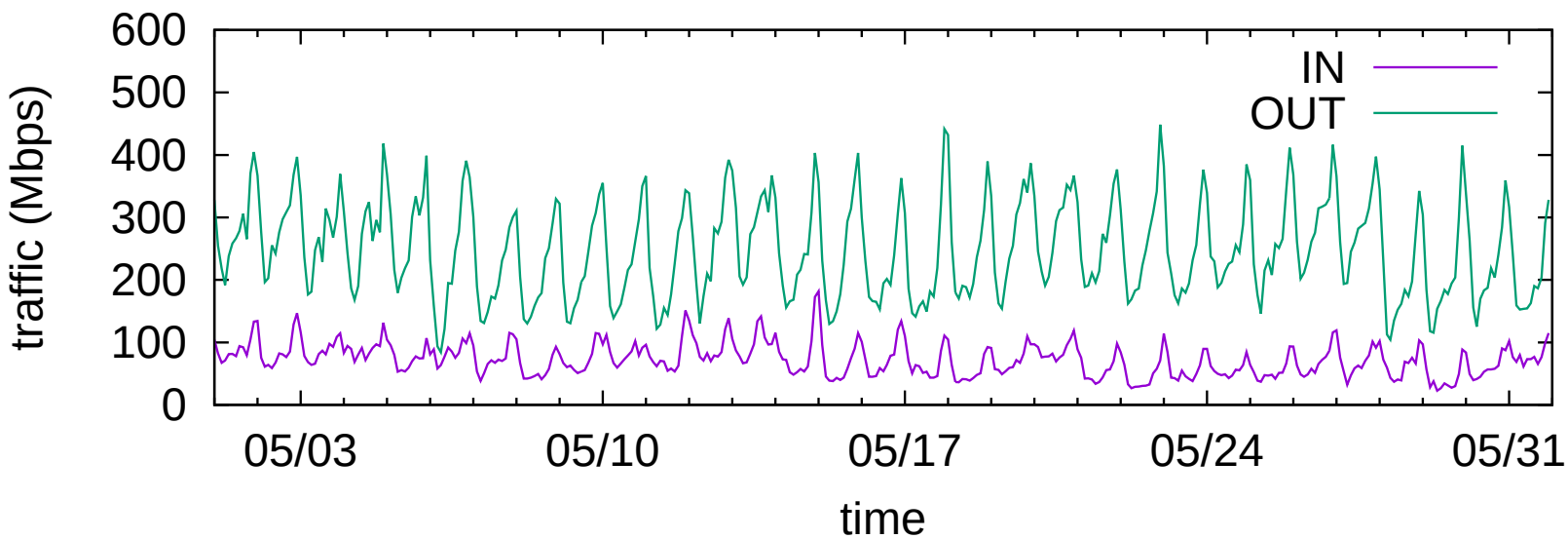
```
set xlabel "timelag k (minutes)"
set ylabel "auto correlation"
set xrange [-100:5140]
set yrange [0:1]
plot "autocorr.txt" using ($1*5):2 notitle with lines
```



今回の演習 2: トラフィック解析

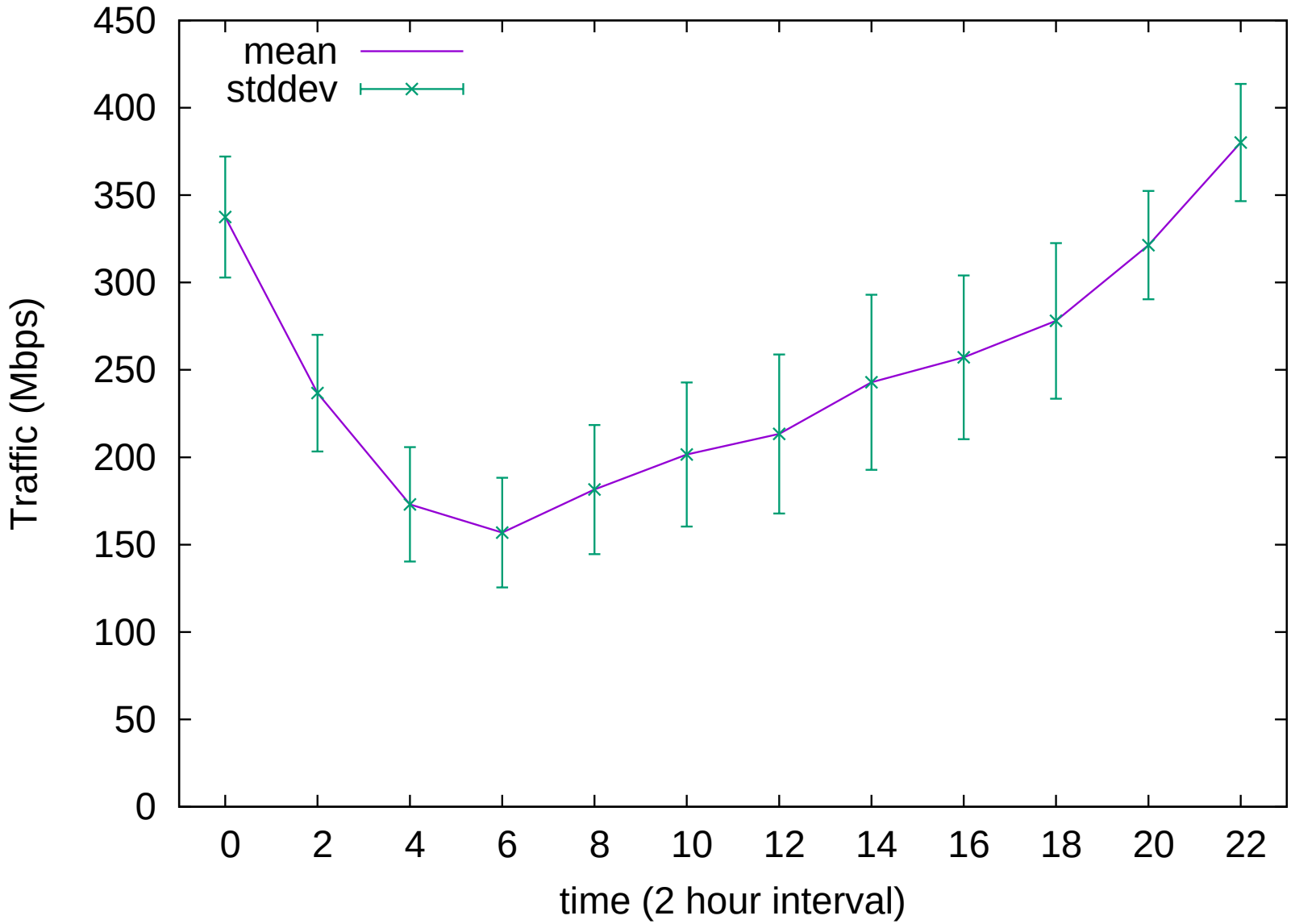
演習用データ: ifbps-201205.txt

- あるブロードバンド収容ルータのインターフェイスカウンタ値
- 2012 年 5 月の 1 ヶ月分、2 時間粒度
- format: time IN(bits/sec) OUT(bits/sec)
- 元データのフォーマットを変換してある
  - original format: unix_time IN(bytes/sec) OUT(bytes/sec)
- ここでは OUT トラフィックを解析
  - IN トラフィックは自習用



今回の演習 2: 時間帯別トラフィック

- 時間毎の平均と標準偏差をプロット



今回の演習 2: 時間帯別トラフィック抽出スクリプト

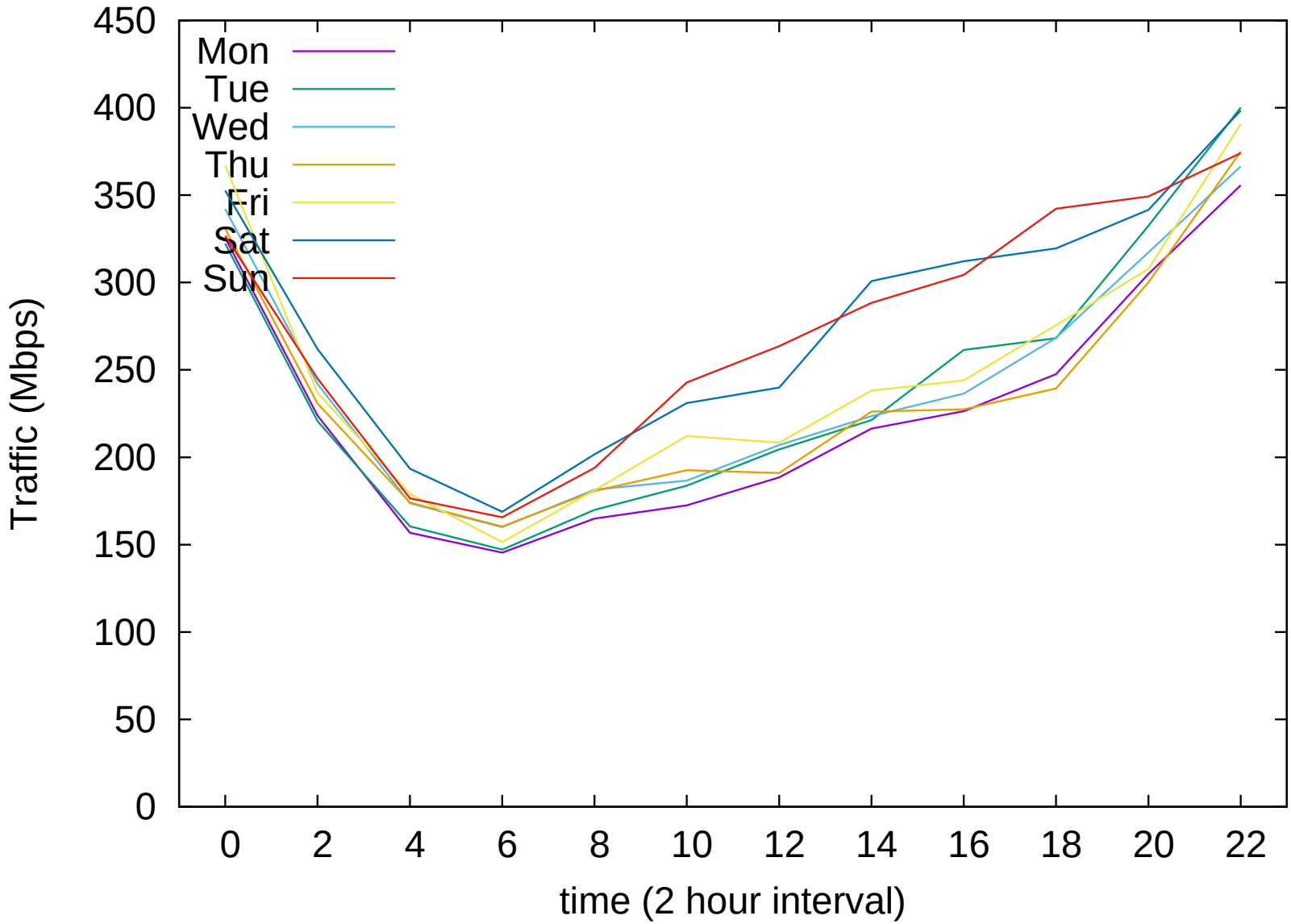
```
#   time  in_bps  out_bps
re = /^\d{4}-\d{2}-(\d{2})T(\d{2}):\d{2}:\d{2}\s+\d+\.\d+\s+(\d+\.\d+)/

# arrays to hold values for every 2 hours
sum = Array.new(12, 0.0)
sqsum = Array.new(12, 0.0)
num = Array.new(12, 0)

ARGF.each_line do |line|
  if re.match(line)
    # matched
    hour = $2.to_i / 2
    bps = $3.to_f

    sum[hour] += bps
    sqsum[hour] += bps**2
    num[hour] += 1
  end
end
printf "#hour\tn\tmean\t\tstddev\n"
for hour in 0 .. 11
  mean = sum[hour] / num[hour]
  var = sqsum[hour] / num[hour] - mean**2
  stddev = Math.sqrt(var)

  printf "%02d\t%d\t%.1f\t%.1f\n", hour * 2, num[hour], mean, stddev
end
```

今回の演習 2: 時間帯別トラフィックのプロット

```
set xlabel "time (2 hour interval)"
set xtic 2
set xrange [-1:23]
set yrange [0:]
set key top left
set ylabel "Traffic (Mbps)"

plot "hourly_out.txt" using 1:($3/1000000) title 'mean' with lines, \
"hourly_out.txt" using 1:($3/1000000):($4/1000000) title "stddev" with yerrorbars
```

今回の演習 2: 曜日別時間帯別トラフィック

- 曜日毎のトラフィックをプロット



今回の演習 2: 曜日別時間帯別トラフィックの抽出

```
#   time  in_bps  out_bps
re = /^\d{4}-\d{2}-(\d{2})T(\d{2}):\d{2}:\d{2}\s+\d+\.\d+\s+(\d+\.\d+)/

# 2012-05-01 is Tuesday, add wdoffset to make wday start with Monday
wdoffset = 0

# traffic[wday][hour]
traffic = Array.new(7){ Array.new(12, 0.0) }
num = Array.new(7){ Array.new(12, 0) }

ARGF.each_line do |line|
  if re.match(line)
    # matched
    wday = ($1.to_i + wdoffset) % 7
    hour = $2.to_i / 2
    bps = $3.to_f

    traffic[wday][hour] += bps
    num[wday][hour] += 1
  end
end
printf "#hour\tMon\tTue\tWed\tThu\tFri\tSat\tSun\n"
for hour in 0 .. 11
  printf "%02d", hour * 2
  for wday in 0 .. 6
    printf " %.1f", traffic[wday][hour] / num[wday][hour]
  end
  printf "\n"
end
```

今回の演習 2: 曜日別時間帯別トラフィックのプロット

```
set xlabel "time (2 hour interval)"
set xtic 2
set xrange [-1:23]
set yrange [0:]
set key top left
set ylabel "Traffic (Mbps)"

plot "week_out.txt" using 1:($2/1000000) title 'Mon' with lines, \
"week_out.txt" using 1:($3/1000000) title 'Tue' with lines, \
"week_out.txt" using 1:($4/1000000) title 'Wed' with lines, \
"week_out.txt" using 1:($5/1000000) title 'Thu' with lines, \
"week_out.txt" using 1:($6/1000000) title 'Fri' with lines, \
"week_out.txt" using 1:($7/1000000) title 'Sat' with lines, \
"week_out.txt" using 1:($8/1000000) title 'Sun' with lines
```

今回の演習 2: 曜日間の相関係数行列

- 曜日間の相関係数行列を計算
  - 曜日間の各時間帯平均値を使う

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Mon	1.000	0.985	0.998	0.991	0.988	0.955	0.901
Tue	0.985	1.000	0.981	0.975	0.969	0.964	0.927
Wed	0.998	0.981	1.000	0.987	0.987	0.946	0.897
Thu	0.991	0.975	0.987	1.000	0.988	0.933	0.859
Fri	0.988	0.969	0.987	0.988	1.000	0.951	0.896
Sat	0.955	0.964	0.946	0.933	0.951	1.000	0.971
Sun	0.901	0.927	0.897	0.859	0.896	0.971	1.000

今回の演習 2: 曜日間の相関係数行列の計算

- 曜日別時間帯別で作った配列を使えばよい

```
n = 12
for wday in 0 .. 6
  for wday2 in 0 .. 6
    sum_x = sum_y = sum_xx = sum_yy = sum_xy = 0.0
    for hour in 0 .. 11
      x = traffic[wday][hour] / num[wday][hour]
      y = traffic[wday2][hour] / num[wday2][hour]

      sum_x += x
      sum_y += y
      sum_xx += x**2
      sum_yy += y**2
      sum_xy += x * y
    end
    r = (sum_xy - sum_x * sum_y / n) /
      Math.sqrt((sum_xx - sum_x**2 / n) * (sum_yy - sum_y**2 / n))
    printf "%.3f\t", r
  end
  printf "\n"
end
```

課題 2: twitter データ解析

- ねらい: 大規模実データ処理の実践
- 課題用データ:
  - Kwak らによる 2009 年 7 月の twitter data、約 4000 万ユーザ分
    - 元データ: http://an.kaist.ac.kr/traces/WWW2010.html
  - twitter_degrees-10000.txt (135KB)
    - 10,000 人分のサンプルデータ
  - twitter_degrees.zip (164MB, 解凍後 550MB)
    - 約 4000 万人分のフルデータ
  - numeric2screen.zip (365MB, 解凍後 756MB)
    - ユーザ ID とスクリーン名のマッピング
- 提出項目
  1. twitter ユーザの following/follower 数散布図プロット
     - 10,000 人分のデータを使った散布図
  2. フルデータによる following/follower 数分布の CCDF プロット
     - X 軸に following/follower 数を取り log-log プロット
  3. フォローワ数の多いトップ 50 ユーザの表
     - ランク、ユーザ ID、スクリーン名、フォロー数、フォローワ数
  4. オプション: その他の解析
  5. 考察: データから読みとれることを記述
- 提出形式: PDF 形式のレポートを SFC-SFS から提出
- 提出〆切: 2015 年 6 月 24 日

課題データについて

twitter_degrees.zip (164MB, 解凍後 550MB)

```
# id followings followers

12      586     1001061
13      243     1031830
14      106     8808
15      275     14342
16      273     218
17      192     6948
18      87      6532
20      912     1213787
21      495     9027
22      272     3791
...
```

numeric2screen.zip (365MB, 解凍後 756MB)

```
# id screenname

12 jack
13 biz
14 noah
15 crystal
16 jeremy
17 tonystubblebine
18 Adam
20 ev
21 dom
22 rabble
...
```

課題 提出物

散布図

- 10,000 人分のデータを用いて、X 軸に following、Y 軸に follower 数を取り log-log プロット

CCDF プロット

- X 軸に following/follower 数を取り log-log プロット

フォローワ数の多いトップ 50 ユーザの表

- ランク、ユーザ ID、スクリーン名、フォロー数、フォローワ数

```
# rank   id           screenname   followings followers

1      19058681       aplusk          183     2997469
2      15846407       TheEllenShow    26      2679639
3      16409683       britneyspears   406238  2674874
4      428333         cnnbrk          18      2450749
5      19397785       Oprah           15      1994926
6      783214         twitter         55      1959708
...
```

sort コマンド

sort コマンド: テキストファイルの行をソートして並び替える

```
$ sort [options] [FILE ...]
```

- options (課題で使いそうなオプション)
  - -n : フィールドを数値として評価
  - -r : 結果を逆順に並べる
  - -k POS1[,POS2] : ソートするフィールド番号 (1 オリジン) を指定する
  - -t SEP : フィールドセパレータを指定する
  - -m : 既にソートされたファイルをマージする
  - -T DIR : 一時ファイルのディレクトリを指定する

例: file を第 3 フィールドを数値とみて逆順にソート、一時ファイルは”/usr/tmp”に作る

```
$ sort -nr -k3,3 -T/usr/tmp file
```

まとめ

第 8 回 時系列データ

- インターネットと時刻
- ネットワークタイムプロトコル
- 時系列解析
- 演習: 時系列解析
- 課題 **2**

次回予定

第 9 回 トポロジーとグラフ (6/15)

- 経路制御
- グラフ理論
- 最短経路探索
- 演習: 最短経路探索